International Journal of Database and Theory and Application
Vol.9, No.2 (2016), pp. 47-60
http://dx.doi.org/10.14257/ijdta.2016.9.2.06

Optimization of GEMV on Intel AVX Processor

Jun Liang[1] and Yunquan Zhang[2]

[1]*Training Center of Electronic Information, Beijing Union University, Beijing, China*
[2]*State Key Lab of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China*
[1]*liangjun2236@sina.com*, [2]*zyq@ict.ac.cn*

Abstract

To improve the performance of BLAS 2 GEMV subroutine under the latest instruction set, Intel AVX, this paper presents a new approach to analyze the new generation instruction set and enhance the efficiency of current data-oriented math subroutines. The whole optimizing process involves memory access optimization, SIMD optimization and parallel optimization. Also, this paper shows the comparison between the traditional SSE instruction set and the AVX instruction set. Experiments show that the optimized GEMV function has obtained considerable increase on performance. Compared with the Intel MKL, GotoBLAS, ATLAS, this optimized GEMV exceeds these BLAS implementations from 5% to 10%.

Keywords*: AVX instruction set, GEMV, BLAS, memory access, SIMD optimization, parallel optimization*

1. Introduction

BLAS (Basic Linear Algebra Subprograms) is the most fundamental and important library of linear algebra. Researchers have conducted many studies on the optimization of BLAS. With the advancement of new hardware platform and software developing tools, optimization of math software libraries should keep up with the pace of computer engineering. The popular implementations of BLAS, such as Intel MKL, GotoBLAS and ATLAS, also have to face the problem and they need to be updated in order to exploit the new hardware features. In reference [1], researchers construct a new method to deal with the matrix in order to improve the locality of reference both in space and time of the data used in the product, which has received great feedback and has been widely accepted by many researchers. In reference [1], researchers attempted to build an automatically tuned BLAS library. Contrary to the traditional optimization method, ATLAS can adjust to the latest platform immediately and it is regarded as the baseline of other optimized BLAS implementations. In reference [2], researchers have made a lot of experiments related with GotoBLAS, Intel MKL, ATLAS under different hardware platforms, which provide a useful guide to choose hardware platform to build large cluster.

Although BLAS has been implemented by a number of hardware vendors and other researchers, the method to maximize the efficacy of hardware platform has little difference to each other. [3]The purpose of this article is to propose an overall approach to optimize the BLAS under the latest Intel AVX micro-architecture and to show the comparison between AVX and SSE instruction sets.

ISSN: 1975-0080 IJDTA

2. Analysis of GEMV Subroutine

2.1. BLAS Level 2 Subroutines

BLAS (Basic Linear Algebra Subroutines) is an important application interface standard in the area of high performance computing, especially for the basic linear algebra operations (such as matrix and vector multiplication). The advent of BLAS bridges the development of high performance algebra computing programs and the specific central computing platform, without developer's knowledge on the hardware architectures and let the developers concentrate on the data and call the functions provided by BLAS. [4]There are three levels in BLAS, each of which execute different functions.

Level 1 contains "Vector – Vector" $(y \leftarrow \alpha x + \beta y)$ and other operations (such as scalar dot products and vector norms)

Level 2 contains "Matrix – Vector" $(y \leftarrow \alpha Ax + \beta y)$ and other operations (such as solving $Tx = y$ for x with T being triangular)

Level 3 contains "Matrix – Matrix" $(C \leftarrow \alpha AB + \beta C)$ and other operations (such as solving $B \leftarrow \alpha T^{-1}$, with T being triangular)

The BLAS Level 2 function GEMV is an important program to compute the product of matrix and vector. The main function of GEMV is to solve the formula with the form as follows

$y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$

In which, A is a matrix and A^T is the transposed matrix of A, x and y are vectors, α and β are scalar numbers.

2.2. Analysis of GEMV Implementation

GEMV is a compute-intensive function, most of the time is used for the exchange of data and operations like add and product. [5] The amount of data is n^2, processing $3n^2$ times multiplication and n^2 times add. The time complexity is $O(n^2)$. Each element in the matrix A is only accessed once, each element of vector X is accessed M times and each element of vector Y is accessed N times. Compared with the GEMM, the reuse rate of matrix A is low, and the transport of data between central process unit and memory is the bottleneck restricting the performance [6].

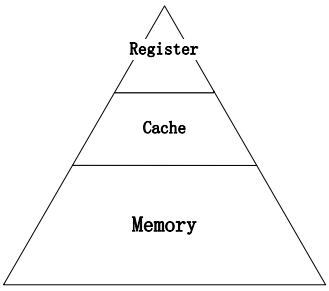

Figure 1. Layout of Memory Hierarchy

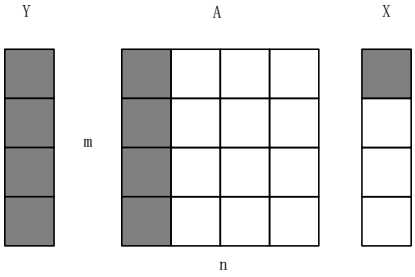

Figure 2. GEMV *m* Blocking

According to the layout of memory hierarchy (Figure1), GEMV has the following implementation algorithms, GEMV *m* blocking, GEMV *n* blocking, GEMV *mn* blocking.

The GEMV m blocking algorithm is shown in Figure2. Assume the matrix A is column major, the A can be streaming read regularly. The algorithm reuses x element one by one, which is read to registers. The vector y is accessed in every loop. If the size of the vector y is smaller than the capacity of the cache, the vector y will be reused at cache level. Thus, this algorithm suits for the small m GEMV case.

The Figure 3 shows the GEMV n blocking algorithm. Unlike the GEMV m blocking algorithm, the matrix A is accessed by a stride m, which results in a bad memory accessing bandwidth. The vector y is accessed one by one, which can be located in registers. If the n is small, the vector x will be reused at cache level. Because of the irregularly accessing the matrix A, we suggests this algorithm is not suitable for column major matrices.

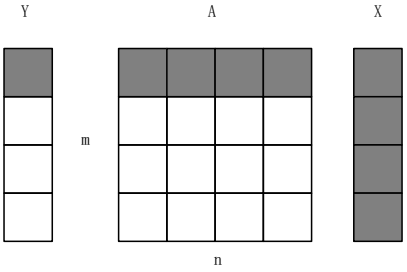

Figure 3. GEMV n Blocking

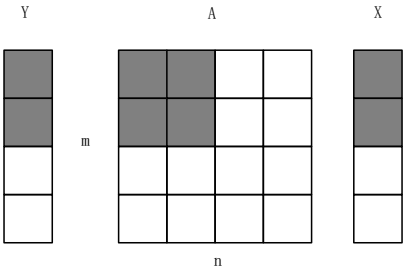

Figure 4. GEMV mn Blocking

The GEMV mn blocking algorithm is a 2D blocking method. We accessed the matrix A by a MbxNb block. Thus, it will reuse the Mb elements of the vector Y and Nb elements of the vector X in the cache.

To sum up, the GEMV m blocking is useful for the small matrix and GEMV mn blocking is suitable for the big matrix.

3. Optimization

The optimization of BLAS 2 functions involves the Intel AVX instruction sets, the bandwidth of the Intel Sandy Bridge CPU, and the memory hierarchy, especially the cache. The main purpose of optimization is to exploit the efficacy of the instruction set, arrange the data flow to reduce the DTLB overhead and LLC miss.

3.1. Hardware Platform

Intel Sandy Bridge micro-architecture is Intel's advanced hardware platform. The main improvement is the powerful ability to compute the float type data, especially for the vectors. These enhancements involve the float algebra calculations, comparison, load, store and others.

The AVX (Advanced Vector Extension) is the instruction set of Intel Sandy Bridge micro-architecture and an extension of the x86 instruction sets. This new generation of instruction set is especially useful to compute the matrix and vector product and guarantees an extraordinary boost up on the matrix calculations.

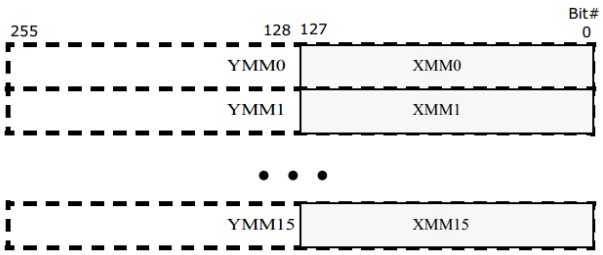

Figure 5. YMM Register and XMM Register

The AVX supports for 256-bit wide vectors and SIMD register set. It supports for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions. With the enhancement of legacy 128-bit SIMD instruction extensions, the AVX instruction set supports three-operand syntax and to simplify compiler vectorization of high-level language expressions.

Intel AVX instruction set extends the traditional x86 SSE family instructions set and can manage the new YMM register, which provide 256 bit float type data computations, as shown in Figure5

3.2. Memory Access Optimization

Instruction set provides developers powerful tools to manage the data from the memory system to the processing unit.[8] However, the method and arrangement of the data flow is also an indispensable part for a high performance program. Only by arranging the data clearly and avoiding overhead can maximize the performance.

Modern computer system must solve the speed gap between the slow main memory and the fast register system. The cache hierarchy is designed to smooth over the huge difference and enhance the performance. In the optimizing process, the cache hierarchy should be analyzed and used carefully to increase the space locality. [10]

Since there is only tiny difference of speed between the L1 and L2 cache and the larger space of L2 cache, the data can be filled into the L2 cache as a chunk. The matrix A are only used once, so the main purpose of cache hierarchy is to increase the reuse of vector X and vector Y. In the experimental system, the L2 cache on Intel Sandy Bridge is 256Kbyte, in other words, the L2 cache can store 215 elements

of double float type. The best way to take advantage of the L2 cache is to access the data chuck in one required dimension of matrix A, rather than the two dimensions.

The main idea of cache hierarchy optimization is to put the whole matrix A into pieces of smaller matrix, the size of which are comparable to the cache, so the machine can access the data in a more efficient way. [11]For the optimization of memory accessing, we selected GEMV mn blocking to implement our GEMV on Intel Sandy Bridge.

```
for(i=0; i<n; i+=Nb)
{
  for(j=0; j<m; j+=Mb)
  {
     GEMV_kernel(Mb,Nb, a, x, y);
     prefetch;
   }
}
```

Figure 6. GEMV mn Blocking Codes

The Figure6 shows the implementation of GEMV mn blocking algorithm. In our experiment, we used different sizes of Mb and Nb to obtain the best solution.

We compute the current block by GEMV_kernel and prefetch the following data.

The prefetching method in the matrix and vectors are also crucial to the memory access optimization. The key problem of prefetch is what to be prefetched, how to determine the prefetch stride, and how frequent prefetch is used by compiler. [9]Although it is better to leave this work to compliers for the modern processors, it can receive some bonus on the performance if it is implemented successfully. The idea on how to manage the prefetch data can be shown in Figure7. The dark gray matrix in the data to be calculated in the current loop, and the light gray matrix is the data needed in the next calculation loop. Because of the limitation of cache line, only half of the light gray matrix is prefetched in the cache. In the optimizing process, experiments are conducted to get the final prefetch solution.

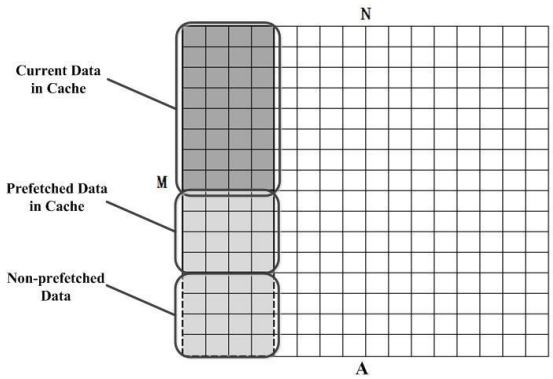

Figure 7. Data Prefetch Flow

3.3. SIMD Optimization for GEMV Kernel

The main operations in the GEMV kernel function are multiplication and add. In addition, there is big traffic load within RAM, cache and register. [7] As a result, the extended instruction set can offer the developer a better solution to the data-

intensive math programs and a faster BLAS can be expected and achieved on this new platform.

The new instructions in Intel AVX are illustrated as follows.

(1) The addpd instruction can add 4 double float number on YMM registers, as shown in Figure8.

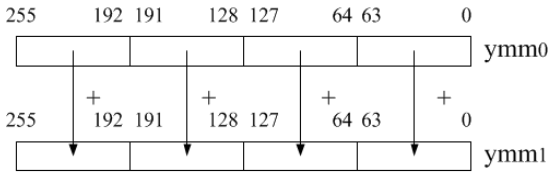

Figure 8. Addpd Instruction

(2) The mulpd instruction can multiply 4 double float number on YMM registers, as shown in Figure9.

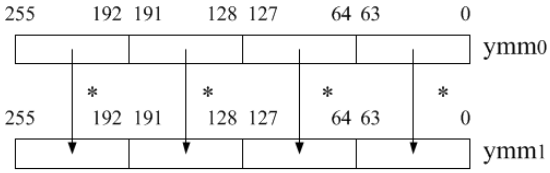

Figure 9. Mulpd Instruction

(3) The vhaddpd instruction can add the float number on two YMM registers' correspond bits onto another YMM register as shown in Figure10.

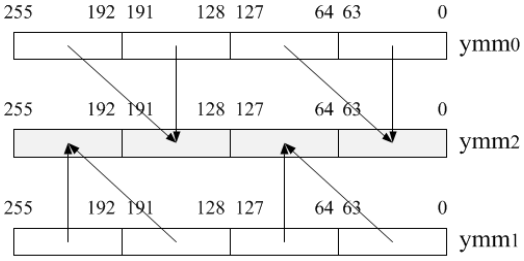

Figure 10. Vhaddpd Instruction

(4) The broadcast instruction can copy the lowest 64 bits double float number to the other three 64-bits positions, as shown in Figure11.

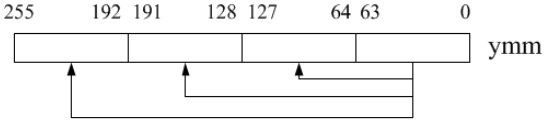

Figure 11. Broadcast Instruction

(5) The prefetch instruction can copy the data which will be used soon close to the cache. There are different prefetch instructions in Intel AVX, as listed in Table 1.

Table 1. Prefetch Instruction

Instruction	Function
Prefetch0	Copy temporary data to all cache
Prefetch1	Copy temporary data to L1 or higher caches
prefecth2	Copy temporary data to L2 or higher caches
prefecthnta	Copy non-temporary data close to processor, avoid the cache pollution

With these new instruction set, the matrix and vectors can be vectorized by 4 double float numbers or by 8 single float numbers.

Our sequential GEMV kernel implementation can be illustrated as following Figure 12.

For the SIMD vectorization of loop j, we choose the step by 4 or 8, which depends on the vector size of the SIMD instructions.

For the loop i, we used loop unrolling technique to optimize the performance. Based on the experiment results, we select the unrolled factor among 2, 4, and 8.

```
//Loop unrolling Loop i
for(i=0; i<n; i+=unrolled_factor)
{
  //SIMD Loop j
  for(j=0; j<m; j+=Vector_Size)
  {
     y[j] += a[j+i*n]*x[i];
     ...
   }
   ...
}
```

Figure 12. GEMV Kernel Implementation

3.4. Parallel Optimization

OpenMP is an ideal tool for the loop parallelization. It supports shared memory multiprocessing programming and can be easily implemented within complier directives, library routines, and environment variables. [12]Without the need of specific knowledge for the programmer on the loop element dependency, OpenMP can control the parallel process and receive great performance increase.

The characters of matrix and vector indicate great potential to be parallelized successfully. As for the GEMV function, after the optimization process mentioned in the article, the sequential implementation can be expressed as following Figure13.

```
for(i=0; i<n; i+=Nb)
{
  for(j=0; j<m; j+=Mb)
  {
     //Unrolling Loop ii
     for(ii=i; ii<i+Nb; ii+=unrolled_factor)
     {
        //SIMD Loop jj
        for(jj=j; jj<j+Mb; jj+=Vector_Size)
        {
            y[jj] += a[jj+ii*n]*x[ii];
          ...
        }
      ...
      }

     prefetch;
   }
}
```

Figure 13. GEMV Sequential Implementation

We can use omp parallel for at the different loops including the outer loop i, loop j, loop ii and the innest loop jj.

For the loop jj case, because of the small length of loop, the workload is light and tiny. Thus, this loop is not suitable for the OpenMP since the overhead of the OpenMP parallel.

For the loop ii case, the workload is bigger than loop jj, which can amortize the overhead of OpenMP parallel. Meanwhile, every OpenMP thread shared the block of matrix A, vector X and vector Y. Thus, this parallel strategy can utilize the last level shared cache at Intel Processors.

For the loop j case, every OpenMP thread compute the sequential MbxNb block in row direction. The workload is bigger than loop ii case. In this parallel method, every thread shares the block of vector x and consumes more cache than loop ii case.

For the loop i case, every OpenMP thread may compute the same block of vector Y. Thus, we need to use the synchronization for the computing correction. Because of the synchronization, this strategy is harmful for the speedup of the parallization.

To sum up, we can parallel the GEMV on loop j and loop ii, which depends on the size of the matrix and the capacity of the cache. For the small matrix, we can parallel the loop j. For the big matrix, we should parallel the loop ii.Because of the big size of testing matrices, we selected to parallel the loop ii.

The next important question in the optimization is how the loop is scheduled in the thread group. [13] OpenMP offers several thread schedule method, static, dynamic and guided. In the parallelization process, the different threads should access the matrix A sequentially to minimize the cache miss, also the vector X and Y should be loaded and stored in the physical order, so the best choice for parallelization is to use the static schedule method.

```
for(i=0; i<n; i+=Nb)
{
  for(j=0; j<m; j+=Mb)
  {
     #pragma omp for schedule(static)
     for(ii=i; ii<i+Nb; ii+=unrolled_factor)
     {
        //SIMD Loop jj
        for(jj=j; jj<j+Mb; jj+=Vector_Size)
        {
            y[jj] += a[jj+ii*n]*x[ii];
          ...
        }
      ...
      }

     prefetch;
   }
}
```

Figure 14. GEMV Parallel Implementation

The Figure14 shows our parallel implementation of GEMV on Intel processors.

4. Performance Evaluation

4.1. Experiment Platform

After the optimization process on the Intel Sandy Bridge micro-architecture, the final optimized GEMV obtains the improvement in both single core and multi core platform. The compiling and testing environment is shown in Table 2.

Since there are two forms of Matrix A, non-transposed and transposed, the GEMV should take advantage of the differences between these two kinds of matrices. The matrix sizes are from 2048 to 10240 by step 1024.

We compared our implementation with Intel MKL 10.3, GotoBLAS2-1.13, ATLAS 3.8.4 version.

Table 2. Experimental Environment

Instruction	*Function*
CPU	Intel Core i7-2600 3.4GHz 4 physical core
RAM	DDR 3 1333 2Gbtye * 2
Operating System	Ubuntu 10.04 LTS 64-bit 2.6.32-39
Compiler	Intel C++ Compiler XE 2011.sp.9.923

4.2. Single-Core Result

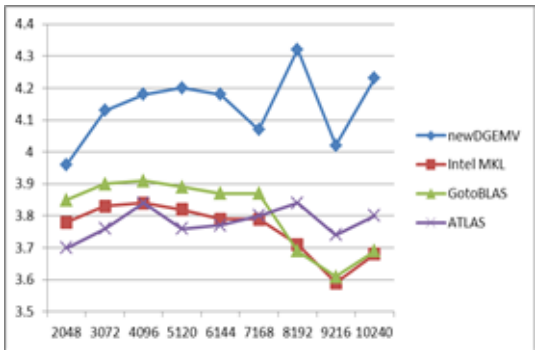

Figure 15. Non-Transposed Matrix Performance on Single Thread

According to the data as shown in Figure15, our implementation (newDGEMV) achieved the best performance, which outperformed other implementation about 8%-10%. The performance of Intel MKL, GotoBLAS, and ATLAS are much closed.

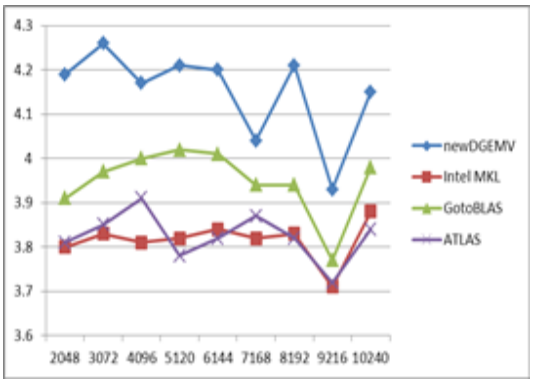

Figure 16. Transposed Matrix Performance on Single Thread

The Figure 16 shows the performance of transposed DGEMV. Our implementation also outperformed other BLAS libraries.

We surpassed GotoBLAS about 5%. Meanwhile, we surpassed the vendor BLAS, Intel MKL, about 10%. We also outperformed ATLAS about 9%.

4.3. Multi-Core Result

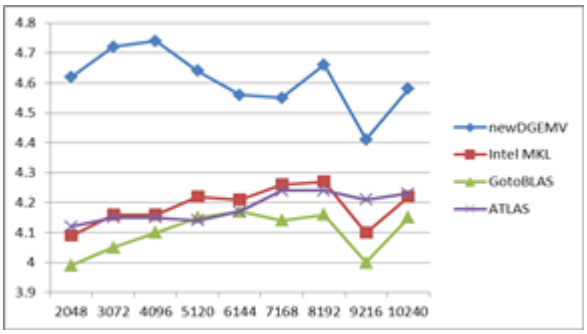

Figure 17. Non-Transposed Matrix Performance on Four Threads

The Figure 17 shows the performance of transposed DGEMV on four cores. Our implementation also outperformed other BLAS libraries.

We surpassed GotoBLAS about 5%. Meanwhile, we surpassed the vendor BLAS, Intel MKL, about 10%. We also outperformed ATLAS about 9%.

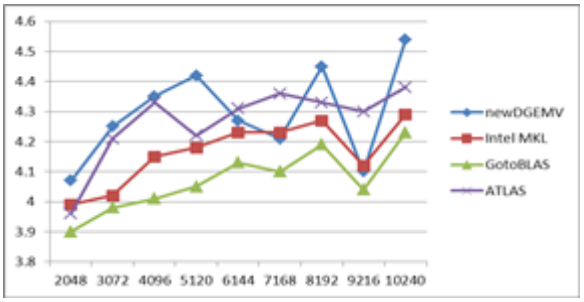

Figure 18. Transposed Matrix Performance on Four Threads

The Figure 18 shows the performance of transposed DGEMV on four cores. Our implementation also outperformed other BLAS libraries.

We surpassed GotoBLAS about 5%. Meanwhile, we surpassed the vendor BLAS, Intel MKL, about 10%. We also outperformed ATLAS about 9%.

5. Related Works

The manual-tuning implementations and auto-tuning implementations are two major efforts to optimize BLAS subroutines on modern processors.

GotoBLAS, which was developed by Kazushige Goto [1], is a famous manual-tuning BLAS library. The GotoBLAS supports various CPU architectures by manually writing different assembly language kernels for them. However, it is no longer under active development. Although OpenBLAS is an effort to continue developing GotoBLAS [15], OpenBLAS doesn't optimize GEMV functions by Intel AVX instructions. Thus, we only compared the performance with GotoBLAS instead of OpenBLAS.

There are many auto-tuning works trying to automatically generate BLAS routines. The ATLAS project was founded by R. Clint Whaley [1], which commits to use empirical techniques to automatically generate portable BLAS library for different architectures. The performance of ATLAS is usually lower than BLAS libraries supported by CPU vendors. According to our test result, our implementation already outperforms ATLAS DGEMV function.

AUGEM is a framework to automatically generating BLAS kernels on x86 CPUs, which supports GEMM, GEMV, AXPY, and DOT subroutines [16]. The AUGEM

framework can apply the user defined templates to the naïve C kernel, which translates to the high optimized assembly language. Thus, it can achieve the similar performance with manual tuning BLAS libraries, including GotoBLAS, and OpenBLAS. However, the AUGEM doesn't adopt OpenMP or Pthread to parallel the functions.

The BTOBLAS compiler can generate the low level C codes from user's similar MATLAB codes [9],[17]-[18]. Via this method, BTOBLAS can compose many BLAS functions into one kernel, which improves the utilization of the cache. Furthermore, the BTOBLAS paralyzes the function by Pthread. While we are focusing on only one GEMV function, the BTOBLAS cannot get the benefit from the composed linear algebra kernels.

The researchers also optimized ATLAS performance on the Loongson MIPS processors, which is the China's homegrown CPU series [5]-[6]. They used the software prefetch instructions and Loongson extended memory accessing instructions to improve the kernels of ATLAS.

6. Summary

In the optimization procedure of BLAS Level 2 subprograms, Intel's latest vector instruction set provide powerful enhancement both on data computing and memory access. The optimized GEMV function has the obtained considerable increase on performance. Compared with the Intel MKL, GotoBLAS, ATLAS, this optimized GEMV exceeds these BLAS implementations from 5% to 10%.

In the testing environment, this optimized GEMV is specialized for data with larger size and the program can take the best of memory bandwidth. On the other hand, for small-size data, the elements in the matrix can be stored directly into the L1 Cache. In this case, there is no frequent data exchange between cache and main memory. Intel MKL and GotoBLAS both have excellent parallel speedup.

Through the optimization process, Intel Sandy Bridge has shown great potential to speed up the data-intensive math functions. However, for functions like GEMV, the Intel AVX instruction set does not improve the performance obviously. On one hand, it is due to the character of GEMV, the bottleneck is resided in the memory bandwidth, which provides the proper traffic loads between the CPU and main memory. [14]Instruction enhancement can only act as the vehicle on the traffic load, but the width is not wider than usual. On the other hand, in the optimization, the memory hierarchy has not been arranged properly, especially on the cache.

In conclusion, this optimized GEMV has obtained performance enhancement of 5 percent to 10 percent on both single thread and multiple thread, compared with the GotoBLAS and Intel MKL. The advanced Intel AVX can be applied to the optimization of fundamental math libraries.

In future, we will investigate the optimization techniques of GEMV and other BLAS 2 subroutines on many-cores processors, including GPU, Intel MIC architecture.

Acknowledgment

This research is supported by the General Program of Science and Technology Development Project of Beijing Municipal Education Commission of China (No. SQKM201411417010). We would like to appreciate all reviewers for their helpful comment on this paper.

References

[1] K. Goto and R. A. V. D. Geijn, "Anatomy of High-Performance Matrix Multiplication", ACM Transactions on Mathematical Software, Article 12, Publication date, vol. 34, no. 3, **(2008)**.
[2] R. C. Whaley, A. Petiet and J. J. Dongarra, "Automated Impractical optimizations of software and the ATLAS project", parallel Computing, vol. 27, no. 1-2, **(2001)**, pp. 3-35.

[3] S. Chen, Y. Zhang, X. Zhang and C. Hao, "Performance test and analysis of BLAS on multi-core processor", Journal of Software, vol. 21, **(2010)**, pp. 214 – 223.
[4] H. Cheng, Y. Zhang, X. Zhang and Y. Li, "Implementation and performance analysis of CPU-GPU parallel matrix multiplication", Computer Engineering, vol. 36, no. 13, pp. 24-29.
[5] B Kågström, P Ling and C. V. Loan, "GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark", ACM Transactions on Mathematical Software, vol. 24, no. 3, **(1998)**, pp. 268–302.
[6] Y. Li, S. He and Q. Li, "Optimization of BLAS 2 on multi-core Loogson-3A", Computer System Application, vol. 20, no. 1, **(2011)**, pp. 163-167
[7] N. Gu, K. Li, G. Chen and C. Wu, "Optimization of BLAS on the Loogson 2F architecture", Journal of University of Science and Technology of China, vol. 38, no. 7, **(2008)**, pp. 854-859.
[8] M. Jiang, Y. Zhang and Y. Li, "Research on the matrix multiplication implementation of GotoBLAS", Computer Engineering, vol. 34, no. 7, **(2008)**, pp. 84-86,103.
[9] S. Byna, Y. Chen and X. H. Sun, "Taxonomy of Data Prefetching for Multicore Processors", Journal of Computer Science and Technology, vol. 24, no. 3, **(2009)**, pp. 405-417.
[10] E. J. Im, "Optimizing the Performance of Sparse Matrix-Vector Multiplication", Computer Science Division (EECS) University of California, Berkeley, **(2000)**.
[11] G. Belter, E. Jessup, I. Karlin, T. Nelson, B. Norris and J. Siek, "Exploring the Optimization Space for Build to Order Matrix Algebra", University of Colorado, **(2010)**.
[12] X. Long, Z. Li and J. Chen, "Impact of optimized BLAS on the performance of parallel program", Journal of Beijing University of Aeronautics and Astronautics, vol. 27, no. 1, **(2001)**.
[13] Y. Li and P. Zhu, "Acceleration Method and Implementation on BLAS", Numeral Computing and Computer Application, no.4, **(1998)**, pp. 227-240.
[14] Z. Chen and A. Storjohann, "A BLAS based C library for exact linear algebra on integer matrices", University Waterloo, Waterloo, ON, Canada, ISSAC 2005, Proceedings of the international symposium on Symbolic and algebraic computation, **(2005)**, pp. 92-99.
[15] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors", SC Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, no. 18, **(2009)**.
[16] Z. Xianyi, W. Qian and Z. Yunquan, "Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor", 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), 17-19 December **(2012)**.
[17] W. Qian, Z. Xianyi, Z. Yunquan and Q. Yi, "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs", In the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13), Denver CO, November **(2013)**.
[18] B. Norris, A. Hartono, E. Jessup, J. Siek, "Generating Empirically Optimized Composed Matrix Kernels from MATLAB Prototypes", ICCS'09 Proceedings of the 9th International Conference on Computational Science: Part I, pp. 248-258.
[19] G. Belter, E. R. Jessup, I. Karlin and J. G. Siek, "Automating the generation of composed linear algebra kernels", SC'09 Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, **(2009)**.
[20] I. Karlin, E. Jessup and G. Belter, "Parallel memory prediction for fused linear algebra kernels", ACM SIGMETRICS Performance Evaluation Review – Special issue on the 1st international workshop on performance modeling, benchmarking and simulation of high performance computing systems (PMBS 10), vol. 38, no. 4, **(2011)**.

Authors

Jun Liang, She was born in Beijing, China, 1962. She received M.S, 2008, and B.S, 1984, from Beijing University of Technology. The major field of study is parallel computing.

Liang has been teaching in Beijing Union University since 1987. Now, she is an associate professor and the director of the New Media Teaching and Research Office. She is the premium member of CCF and chapter member of SAMSS. Her research interests include Large-scale data index, compression and management. She has published more than 20 articles on the domestic core journals and international prestigious conferences. She has rich experience in large-scale matrix parallel computing and in-depth understanding of cutting-edge technology about image parallel processing.

Yunquan Zhang, He was born in Shandong, China, 1973. He received Ph.D. from Graduate School of Chinese Academy of Sciences, 2000, and B.S. from Beijing Institute of Technology, 1995. The major field of study is parallel computing.

He is a Professor at the State Key Lab. Of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China. His research interests include parallel algorithm and parallel software, parallel computing model, performance benchmarking and optimization.

Prof. Zhang holds the positions of General Secretary of SAMSS and General Secretary of CCF HPCTC. He received second class National Award on Science and Technology Advancement on 2000.

